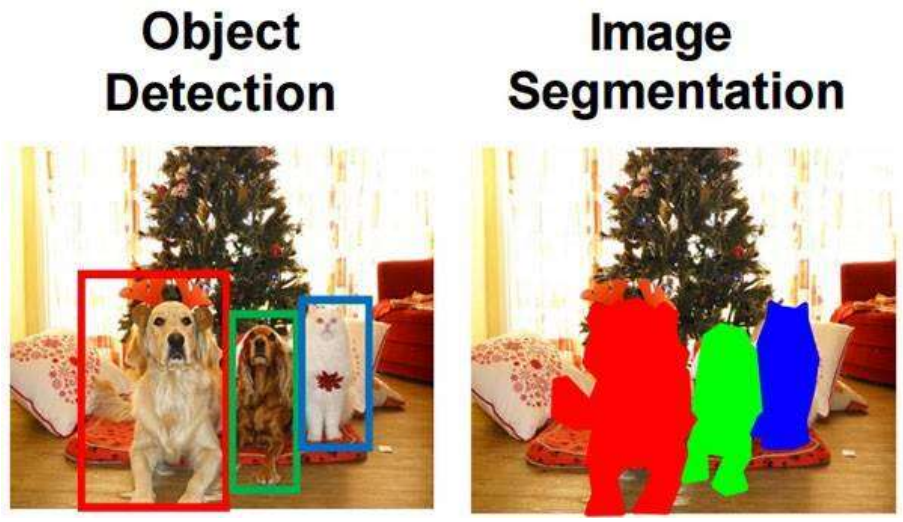
Image Segmentation vs Object Detection

In computer vision, Image Segmentation is the process of partitioning an image into multiple segments (each segment is a set of pixels which represents an image object) . To be clear, Image Segmentation is the concept behind the self-driving car which allows it to recognize the road, pedestrians and other cars at a pixel-level accuracy. Differently from Object Detection, which is merely about detecting an object in an image drawing a square box around it, Image Segmentation consists of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.



As you can see from the image above Object Detection doesn't tell us anything accurate about the exact shape of the object detected whilst the Image segmentation creates a pixel-wise mask for each object in the image, giving us a more granular understanding that is fundamental for some real-world problem:

- Medical diagnosis (i.e. cancer detection where the shape of the cancerous cells is the major indicator for establishing the severity of the illness),
- Locating objects using satellite images,
- Self-Driving Cars

Image Segmentation techniques

Four different Image Segmentation techniques are going to be analysed :

1. Region-based Segmentation
2. Edge Detection Segmentation
3. Cluster-based Segmentation
4. Mask R-CNN

Region-based Segmentation

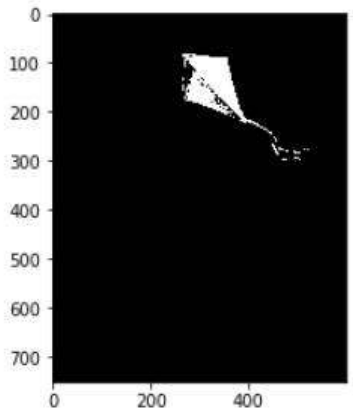
The simplest way to segment different objects could be using the pixel values.
For example consider to have an image containing a red kite in a blue clear sky,

all we need to do is setting a **global threshold value**. The pixels values falling below or above that threshold can be classified accordingly (i.e. foreground or background). There are multiple ways to set a threshold value (i.e. using the mean value and so on), but in this occasion it was considered appropriate to set the threshold value = 0.3

```
gray = rgb2gray(img_P)
gray_r = gray.reshape(gray.shape[0]*gray.shape[1])

for i in range(gray_r.shape[0]):
    if gray_r[i] > 0.3:
        gray_r[i] = 0
    else:
        gray_r[i] = 1
gray = gray_r.reshape(gray.shape[0],gray.shape[1])
plt.imshow(gray, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x1fa98e029b0>
```



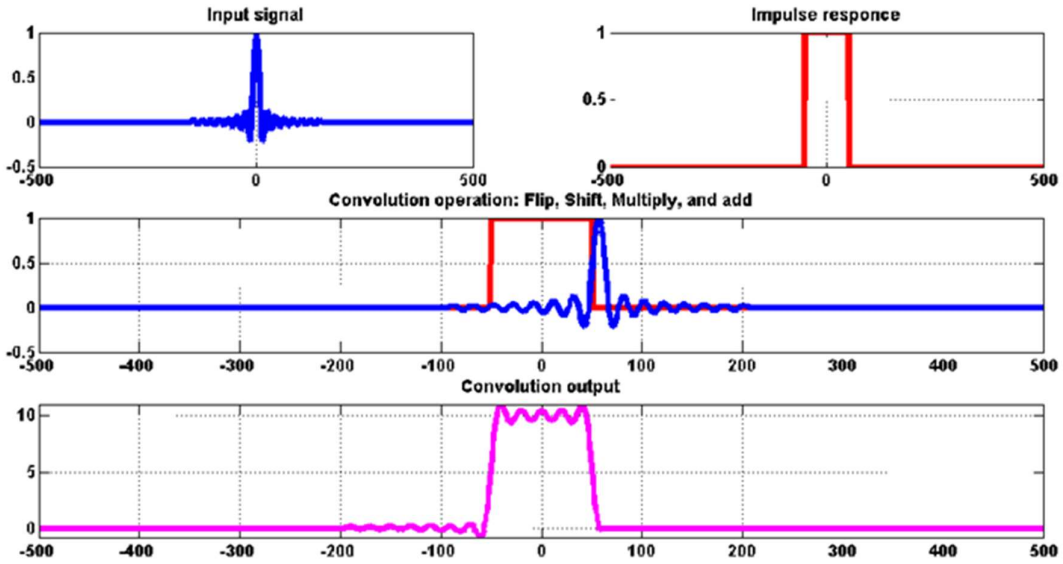
The above code contains the code used to assign the label to every pixel based on its value. We convert the image from RGB to grayscale just for simplicity. How is easily understandable this

method works if there is a sharp contrast between the objects and the image's background. If we want to divide the image into two regions (object and background), we define a single threshold value and this is known as the **Global Threshold.** If we want multiple objects along with the background, we must define multiple thresholds and these thresholds are collectively known as the **Local threshold.**

Edge Detection Segmentation

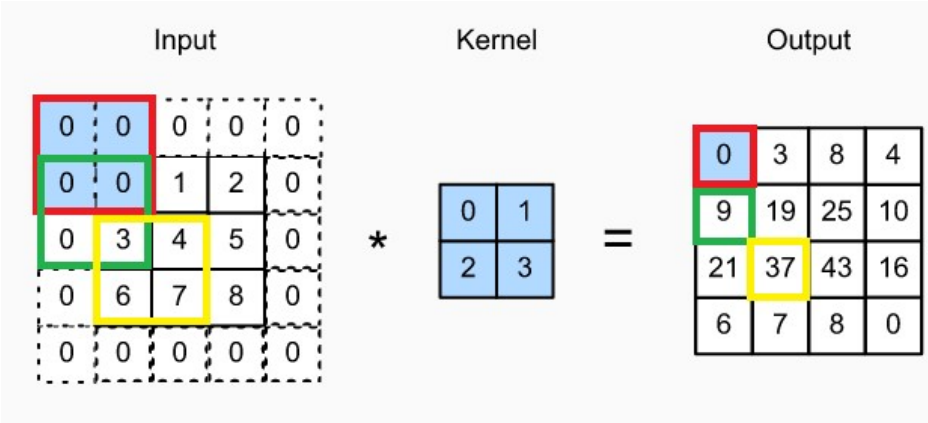
The points at which image brightness changes sharply are typically organized into a set of curved line segments termed *edges* or, more formally, has discontinuities. So an edge can be defined as a significant local changes of intensity in an image. We can make use of this particular property to detect edges in order to define a boundary of the object and to detect the shape of multiple objects present in an image.
The mathematical operation used to detect is the convolution.

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. It provides a way of `multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality. More formally convolution is a mathematical operation that takes in input two functions and as the output produces a third function expressing how the shape of one is modified by the other one.



In Image Segmentation The convolutional operation is performed using matrices called kernels. Specifically a kernel is a square matrix that is applied to the image in order to emphasize certain features or remove some other ones. **Remember that before performing the convolution operation a kernel has to be flipped with respect to the axis along which you are performing the convolution. If it didn't flip it would be correlation instead of a convolution.**

So the question is how a kernel convolves over an image :



*Stride = Denotes how many steps we are moving between applications of the kernel (by default is one).

*Padding = It is about adding extra pixels outside the image

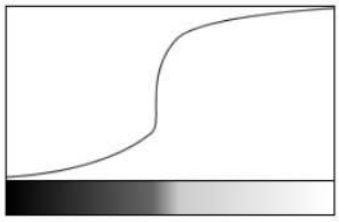
Red_square = > 0 * 0 + 0 * 1 + 0 * 2 + 0 * 3 = 0
Green_square = > 0 * 0 + 0 * 1 + 0 * 2 + 3 * 3 = 9
Yellow_square = > 3 * 0 + 4 * 1 + 6 * 2 + 7 * 3 = 37

In the below example (*stride = 1, No padding), we begin with the convolution window positioned at the top-left corner of the input array and slide it across the input array, both from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray contained in that window and the kernel array are multiplied (elementwise) and the resulting array is summed up producing a single scalar value. This result gives the value of the output array at the corresponding location.

In order to detect the edges we will be using two particular types of kernels called **Sobel Filters** and **Laplacian Operator.**

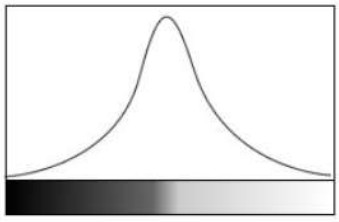
- The Laplacian Operator is a measure of the 2nd spatial derivative (spatial derivative refers to how much the image intensity values change per change in image position) of an image. Because we are using a 2nd derivative the Laplacian operator is extremely sensitive to noise, normally something like Gaussian blur is used as well in order to reduce it.
- The Sobel Filters work with first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes. it is less affected by noise but generally the results are worse than the Laplacian Operator

Because sometimes I find really useful to visualize some mathematical concepts for a better understanding I will add some graphs that can easily explain how these two kernels work



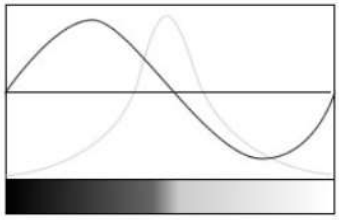
The curve representing intensity

Imagine to have a 1-D image where the intensity of each pixel is plotted as a graph. The Darker is the bit the lower is the intensity (y-axis)
At the centre the curve has a steep slope, meaning you got an edge



The first derivative of the curve above

The Sobel filters use the first derivative so if the peak is steep enough it represent an edge pixel.
Imagine you have multiple peaks of different intensity how to decide which peak represents an edge and which does not ?
A kind of threshold should be set to classify all peaks above as an edge pixel else it must be considered part of noise.



The second order derivative

We know that the second derivative tells us how fast the change is changing. On the left the slope is positive so the curve is rising. On the right, the slope is negative.
There must exist a point where there is a zero crossing. That point is the edge's location. This idea is used by the Laplacian edge detectors.

In mathematical terms the Laplacian L(x,y) of an image with pixel intensity values I(x,y) is given by :

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Coming back to our problem, an image is a set of discrete pixels so we have to find a convolutional kernel that can approximate the second derivatives in the definition of the Laplacian operator. Two commonly small kernels are :

0	−1	0
−1	4	−1
0	−1	0

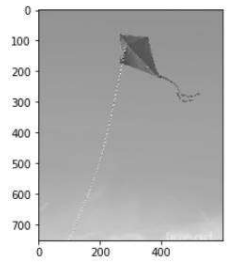
−1	−1	−1
−1	8	−1
−1	−1	−1

Now let's see an example of a Laplacian operator application using the image already used previously, the one with the kite.

```
from scipy import ndimage

kernel_laplace = np.array([np.array([1, 1, 1]), np.array([1, -4, 1]), np.array([1, 1, 1])])
out_l = ndimage.convolve(gray, kernel_laplace, mode='reflect')
plt.imshow(out_l, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x16f11671b00>
```



The input is a grayscale image and applying the Laplacian operator we are able to detect all edges (If you do not believe me just write down all multiplications print out the resulting array as an image and you will get a confirmation 😊 😊).

Cluster-based Segmentation

This technique comes from the idea that we can segment the image using clustering algorithms like k-means. The pixels belonging to each cluster will share the similar characteristics and so will be classified within the same group. The key advantage of using a clustering algorithm like k-means is that it is very simple and easy to understand.

Let's try this technique on this input image :

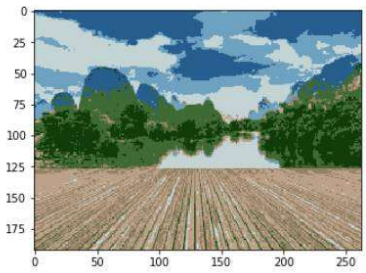
```
# It's a 3-dimensional image For clustering the image using k-means,
# we first need to convert it into a 2-dimensional array whose shape will be (length*width, channels)
pic_n = pic.reshape(pic.shape[0]*pic.shape[1], pic.shape[2])
pic_n.shape
```

```
(50496, 3)
```

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=7, random_state=0).fit(pic_n)
pic2show = kmeans.cluster_centers_[kmeans.labels_]
```

```
cluster_pic = pic2show.reshape(pic.shape[0], pic.shape[1], pic.shape[2])
plt.imshow(cluster_pic)
```

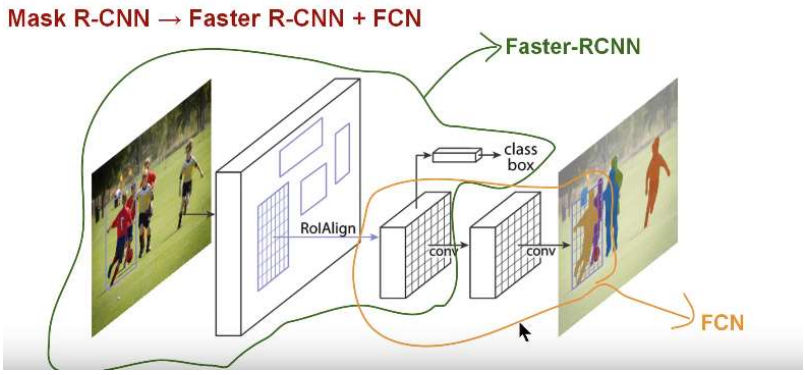
```
<matplotlib.image.AxesImage at 0x13cd7f61438>
```



Mask R-CNN Segmentation

Mask R-CNN is a deep learning method created by Facebook researchers that is able to create a pixel-wise mask for each object in an image. This method extends Faster R-CNN by adding a mask branch used for predicting segmentation masks on each Region of Interest (RoI), in parallel with the existing branch for classification and bounding box regression. Specifically, this mask branch is a small FCN (Fully Convolutional Network).

<u>This technique is the current state-of-art for image segmentation and runs at 5fps</u>



In order to give a neat description of how this whole system works every bit will be explained separately :

1. **Faster R-CNN**
    a. **ResNet101**
    b. **RPN (Region Proposal Network)**
    c. **ROI Align (Region of Interest)**

2. **FCN (Fully Convolutional Network)**

Faster R-CNN

Faster R-CNN represents the state-of-art of object detection. Receiving an image in input it uses a backbone* model (ResNet101 for the best performing Mask R-CNN) to extract the features, next this feature map is passed to a Region Proposal Network that, as the name revealed, is a network which creates “proposal” for the region where the object lies, then ROI is responsible for producing the same-size feature maps from non-uniform inputs received from the RPN network.

*backbone = In deep learning backbone is a term used to refer to the feature extractor network

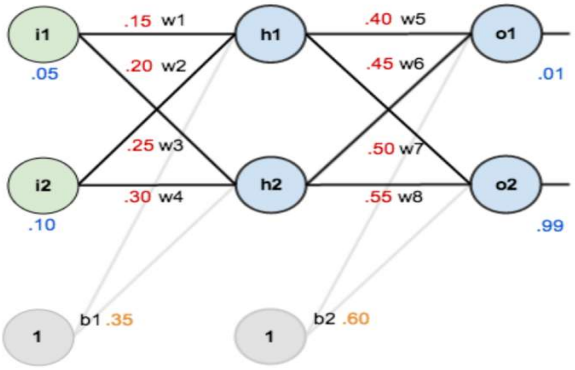
ResNet101

This is the part that probably all of you knows already because ResNet101 or more in general ResNet (Residual Network) is one of the most famous deep learning architecture and algorithm used for computer vision problem.
It is well known that a deep neural network can be very difficult to train in terms of time resources and accuracy. A Residual neural network is a type of network typically very deep (ResNet101 has 101 convolutional layers) which ease the training part utilizing mechanism like *skip connection* or *shortcuts* to jump over some layers, typical ResNet models are implemented with double or triple layer skips. Specifically skipping connection avoids the problems of **vanishing/exploding gradients** and **time complexity** that are very likely to occur when we are dealing with a deep neural network.

Vanishing/Exploding gradients complexity problem

The vanishing/exploding gradients is an issue that occurs during training machine learning algorithm through gradient descent. Often occurs in deep neural networks, because of the property of being "deep". During backpropagation (backpropagation algorithm is used to update the weights in order to improve the performance of the network reducing the error), you have to calculate the partial derivatives of the neurons in order to update the weights. The gradients coming from the deeper layers have to go through continuous matrix multiplications because of the chain rule, and as they approach the earlier layers, if they have small values (<1), they shrink exponentially until the vanish and make it impossible for the model to learn, this is the vanishing gradient problem. While on the other hand if they have large values (>1) they get larger and eventually blow up and crash the model, this is the exploding gradient problem.

----------**BACKPROPAGATION EXAMPLE**----------



Forward step (Calculating the output) :

-**Calculate the input for the neuron** h1 : $net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1 = 0.3775$

-**We squash the input using use the logistic function in order to generate the output** h1 :
$out_{h1} = \frac{1}{1+e^{-ne\ h1}}$ = 0.5932

- "***doing the same thing for h2***" ... $out_{h2} = 0.5968$

-Calculate the input for the neuron o1 : $net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$ = 1.1059

-Calculate the output o1 : $out_{o1} = \frac{1}{1+e^{-net_{o1}}}$ = 0.7513

-"***doing the same thing for o2***" … out_{o1}= 0.7729

-Calculating the Total Error : $E_{total} = \sum \frac{1}{2}(target - output)^2 = 0.2748$

Backward step (Update the weights) **:**

Calculating partial derivative for w1 using the chain rule :

$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial wi}$ => after all the calculations = 0.00043

$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{01}}{\partial ou_{h1}} + \frac{\partial E_{02}}{\partial out_{h1}}$

$\frac{\partial E_{01}}{\partial out_{h1}} = \frac{\partial E_{01}}{\partial net_{o1}} + \frac{\partial net_{o1}}{\partial out_{h1}}$

$\frac{\partial E_{01}}{\partial net_{o1}} = \frac{\partial E_{01}}{\partial out_{o1}} + \frac{\partial out_{o1}}{\partial net_{o1}}$

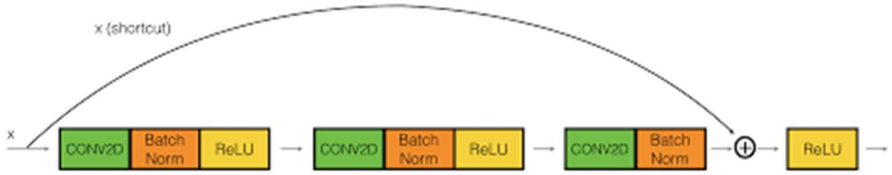
$w_1 = w_1 - \eta \frac{\partial E_{total}}{\partial w_1}$ = 0.15 – 0.5 * 0.00043 = 0.1497 => w1 is updated

Chain rule : Turns a complicated derivative into several easy onesð z / ð x = (ð z / ð y) * (ð y / ð x)

Updating the weights will help us to reduce the error improving the network performance eventually.

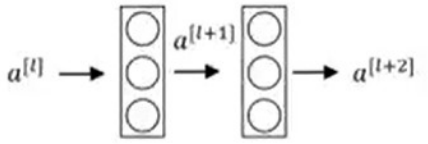
Now imaging to have dozen of neurons, all the multiplications between small numbers (**i.e. to update w1**) will produce an even smaller number close to 0 that won't be able to improve the performance (**vanishing gradient**) or if we have large values when we update **w1,** its value will increase exponentially (**exploding gradient**)

To solve the problem of vanishing/exploding gradients a skip connection is added :



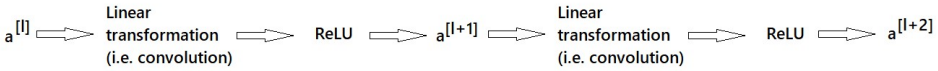
How this skip connections works ? Let's see an example

Normal flow in a neural network

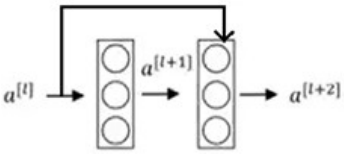


Where :
a = activation function,
l = layer
z = weighted sum
b = bias
g = ReLU Function
W = weights

$$z^{[l+1]} = W^{[l+1]} * a^{[l]} + b^{[l+1]}$$
$$a^{[l+1]} = g\left(z^{[l+1]}\right)$$
$$z^{[l+2]} = W^{[l+2]} * a^{[l+1]} + b^{[l+2]}$$
$$a^{[l+2]} = g\left(z^{[l+2]}\right)$$

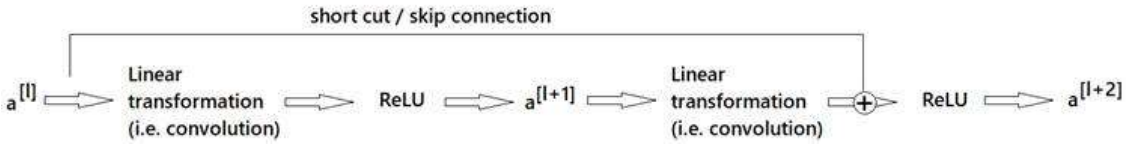


Skip connection flow



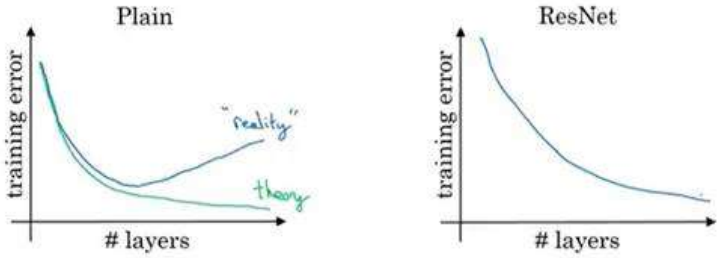
In this way we skip the first block :

$$a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$$



Skipping the layer we are able to pass the information deeper into the neural network. The addition of $a^{[l]}$ makes the **residual block.**
The inventor of ResNet found out that using these residual block allows you to train much deeper neural networks.

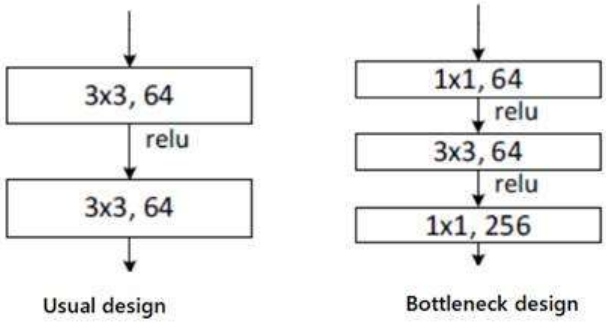
Difference in performance



As the number of layers increases a plain network without any skip connection would have a quadratic error function in reality as opposed to ResNet

Time complexity problem

Since a network can be very deep, the time complexity is high. A bottleneck design is used to reduce complexity :



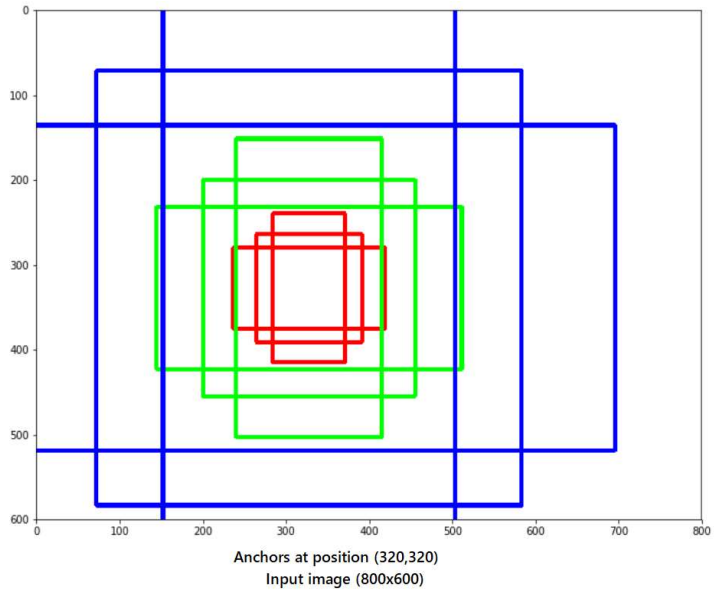
This is a technique suggested in GoogleLeNet. In order to reduce the training time, a stack of 3 layers (**bottleneck design**) is used instead of 2. The three layers are 1 x 1, 3 x 3 and 1 x 1 convolutions, where the 1 x 1 layers are responsible for reducing and then restoring dimension. It turns out that 1 x 1 conv can reduce the number of connections while not degrading the performance of the network so much. With the bottleneck design ResNet34 becomes ResNet50, and there are deeper network with the bottleneck design ResNet101, ResNet152

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

RPN (Region Proposal Network)

This network has the task of generating region proposals, that will be used to detect objects. RPN generates a number of proposals that will be filtered, just keeping those ones most likely containing objects.

The most important concept to bear in mind is ***The Anchors .*** An anchor is simply a box that is built using a pixel as landmark. In the default configuration to Faster R-CNN, 9 anchors are generated for each position:

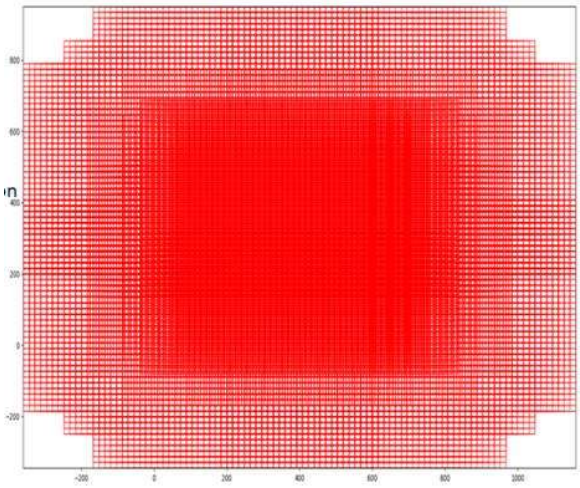


Anchors at position (320,320)
Input image (800x600)

Explanation :

- The three different colours represent three scales or sizes : 128x128, 256x256, 512x512
    - For each of this colour boxes/anchors with three different height width rations namely 1:1 1:2 and 2:1
    - Doing some calculations we will have Height * Width* 9 (anchors)

An input image of 800x600 after placing all the anchors boxes would appear in this way :



ll these anchors are run through two (1x1) kernel convolutional:

- The first 1x1 convolutions layer will produce the background/foreground class scores (the probability that an anchor contains an object or it's just background).
  To generate the labels background or foreground IoU (Intersection over union) of all the bounding boxes against the ground truth boxes are taken. In simple words
  An anchor is considered to be a "positive" (foreground) if it satisfies two conditions: the first one is that the anchor has an IoU greater than 0.7 with any ground truth box, the Second condition is that the anchor has the highest IoU with a ground truth box. An anchor is labelled "negative" (background) if its IoU with any ground truth boxes is less than 0.3.
  N.B. The remaining boxes are disregarded during the training phase. Furthermore at train time all the anchors that cross the boundary are ignored so that they do not contribute
  Negatively to the loss.
  After these two steps the remaining boxes are retained and clipped to image bondary eventually
- The second one is used for determining the coordinates of our "positive" boxes. We need to learn the offsets of the anchor box coordinates (x,y,w,h) respecting the ground truth box , where (x,y) is the centre of the box, w and h are width and height. For learning these offsets we need to have targets. These targets are generated by comparing the anchor boxes with ground truth boxes in a process called "anchor target generation". This process calculates the difference in the coordinates, between the ground truth and the anchor box as target to be learned by regressor. The output of this regressor is represented by 4 coefficients of each of the 9 anchors for every point in the feature map. These regression coefficients are used to improve the coordinates of the anchors that contain objects

The RPN training loss is given by :

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*).$$

Where:

i : is the index of the anchor in the mini-batch,
$L_{cls}(p_i, p_i^*)$: is the classification loss represented by the log loss over two classes
p_i : is the output score from the classifiation branch for anchor i
p_i^* is the groundtruth label (1 or 0)
$L_{re}(t_i, t_i^*)$ is the regrssion loss and is activated only if the anchor is positive (p_i^* = 1)
t_i is the output rediction of the regrssion layer and consists of the 4 predicted coordinates of the bounding box
t_i^* consists of 4 coordintes that roughly represent how far the anchor is from the groundtruth box that the anchor overlap

ROI Align (Region of Interest)

Before explaining ROI Align it is necessary to give a brief introduction about ROI Pooling.
ROI Pooling it is a type of max pooling applied to all proposals from the backbone feature map (the output of RPN) in order to produce the fixed-size feature maps from non-uniform inputs.

ROI Pooling layer takes two inputs:

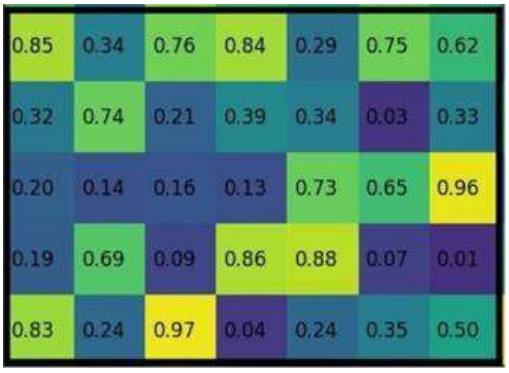
- A feature map obtained after multiple convolutions and pooling layers
- An N X 5 matrix containing the list of regions of interest, where N is a number of ROIs and each row appears as follows : [index of feature map, x,y,w,h]

So for every region of interest in the input matrix, ROI pooling takes a section of the input feature map that corresponds to it and scales it to some pre-defines size (i.e. 7x7).
The scaling is done by:

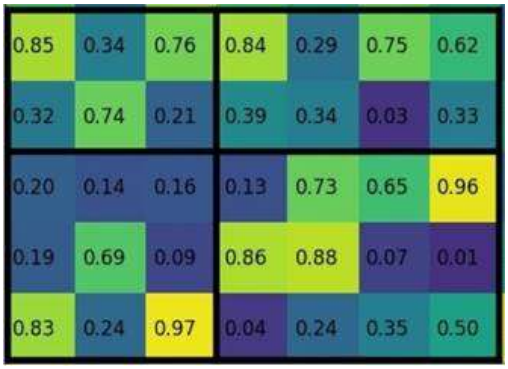
1. Dividing the region proposal into equal-sized sections (number of which is the same as the dimension of the output)
2. Finding the largest value in each section
3. Copying these max values to the output buffer

Example :

Considering a region of interest of 7x5 and an output size of 2x2.



So as we said we have to divide this region into equal-sized section bearing in mind the output size.

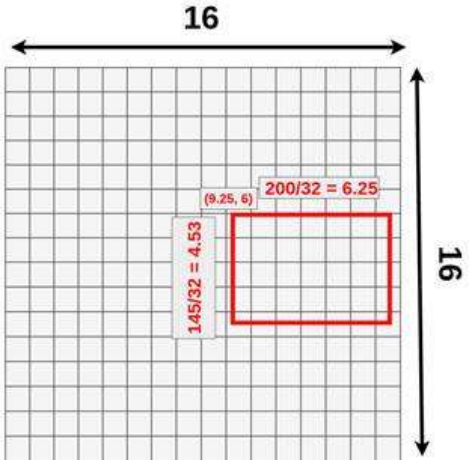


We take the highest value of each section and it represents the output from the region of interest pooling layer.

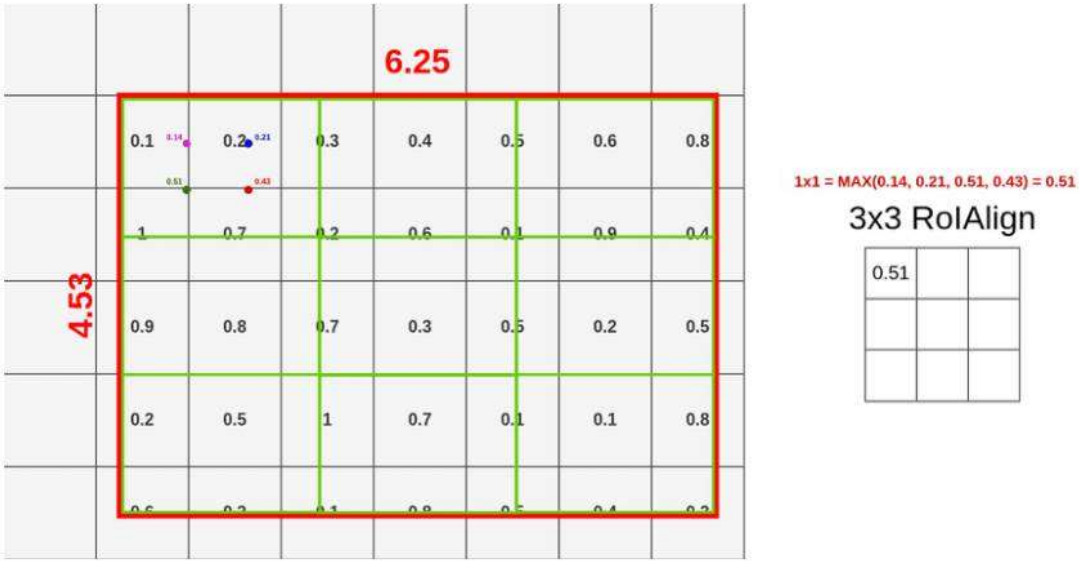
Although the ROI pooling works quite well, it helps us significantly speed up both training and test time and let us reuse the feature map, there is a small problem that may lead to performance degradation.

Imaging to have an image input of size 512 X 512 X 3 used to generate all **anchor boxes** and after all convolutional layers the input is mapped into a 16 x 16 x 512 feature map, so we have a scale factor of 32.
Next, we are trying to map one proposed region of interest 145 x 200 box onto the feature map. Because 145 and 200 cannot be divided exactly by 32 and ROI pooling is applying quantization, we can lose piece of information that can deteriorate the model performance.
ROI Align is designed to fix the location misalignment caused by quantization in the RoI pooling, by using bilinear interpolation for computing the floating-point location values in the input.

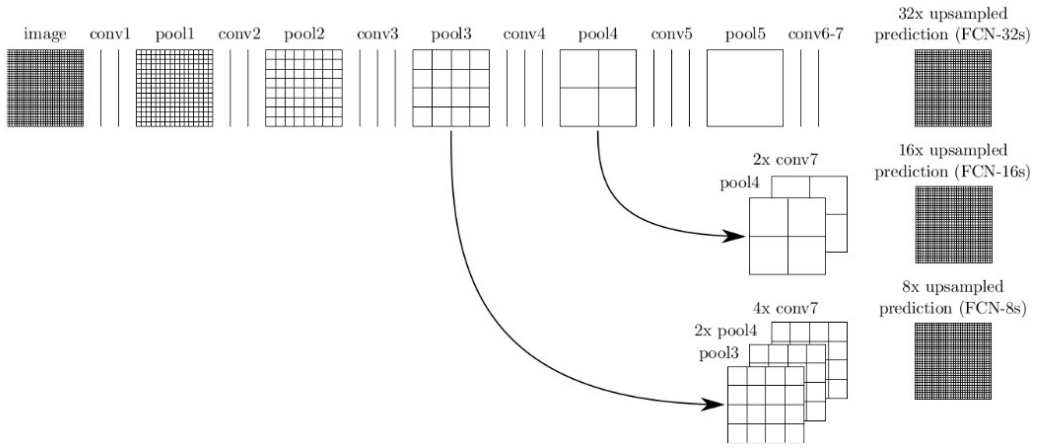
Graphic example



Classic ROI pooling would have considered a 4x6 window, discarding the decimal part



- Down sampling path : capture semantic / contextual information
- Up sampling path : recover spatial information



- conv layer is represented by a vertical line
- pool represents a pooling layer

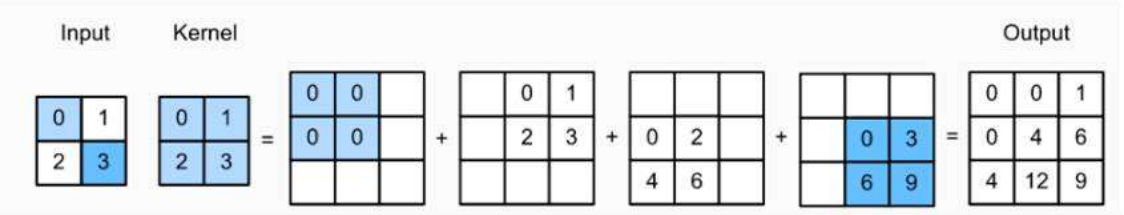
Three different architectures are shown in the figure above, namely:

- FCN-32 : Directly produces the segmentation map from conv7, by using a transposed convolution layer with stride 32
- FCN-16 : Sum the 2x up sampled prediction from conv7(using a transposed convolution with stride 2) with pool4, then produces the segmentation map by using a transposed convolution layer with stride 16 on top of that
- FCN-8 : Sums the 2x up sampled conv7 (with a stride 2 transposed convolution) with pool4, up samples them with a stride 2 transposed convolution and sums them with pool3, and applies a transposed convolution layer with stride 8 on the resulting feature maps to obtain segmentation map

Up sampling via Transposed convolution

Convolution is a process getting the output size smaller.
Transposed Convolution (also known as Deconvolution which is not appropriate because it implies removing the effect of convolution which we are not aiming to achieve) are used to up sample the input feature map to a desired output feature map using some learnable parameters.

Example:



Consider a 2x2 feature map in input which need to be up sampled to a 3x3 feature map using a 2x2 kernel.

What we have to do is take every element of the input multiplying it with every element of kernel and sum

Input[0,0] * kernel = [[0,0],[0,0]]
Input[0,1] * kernel = [[0,1],[2,3]]
Input[1,0] * kernel = [[0,2],[4,6]]
Input[1,1] * kernel = [[0,3],[6,9]]

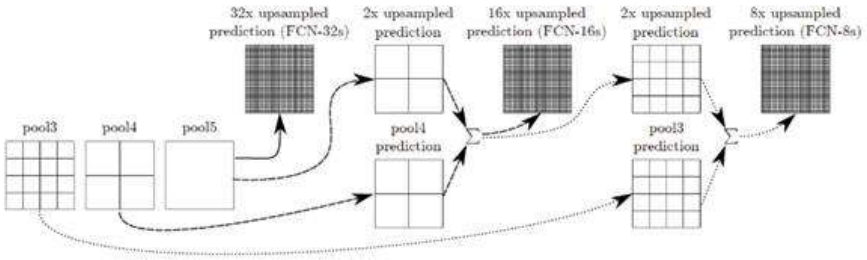
So now we understood how to up sample the output of a convolution block (pooling layer), but why we should do it and why a sum is mentioned when different FCN architectures are explained ?

Because this is a instance segmentation problem we have to up sample in some way the output after all convolution operations to make it to have the same size of input image. After going through conv7 as in the FCN architecture image, the output size is small so we up sample by a factor of 32. The problem is that after several convolution blocks, useful to get deep features, spatial location information are lost. To fix this problem we are going to fuse (by element-wise addition) the output from deeper layers with those ones from shallower layers which contain more spacial information. This fusing operation it's like boosting/ ensemble technique, in which we add output from different layers to improve the prediction

Output from different convolutional layers have different size, so we need to up sample them in order to sum.

FCN-32 = The output from pool5 is 32x up sampled
FCN-16 = The output from pool5 is 2x up sampled fuse with pool4 then 16x up sampled
FCN-8 = The output from pool5 is 2x up sampled fuse wit pool4, 2x up sampled and fuse with pool3

Bit confusing? The image below should clarify the concept better



To conclude let's see an image segmentation output using the three different architectures, FCN-32, FCN-16 AND FCN-8